TEXTRACT

a demiliterate* programming system

Fr. Saul N. Braindrane†
Bernard Alan Sufrin‡

April 1st, 2008
(revised December 2020, and August 2021)

Abstract

You were convinced by the literate programming propaganda, and you wanted to be able to write an article about a program, and to derive the article and the text of the program from the same manuscript. So you went out there on the Web and read all you could about the literate programming systems that were around and got that sinking feeling.... It may be time to consider demiliterate programming.

> "Demiliterate programming is to literate programming as ping-pong is to chess."[1(p19)]

*pronounced *demmy literate*

†Saul Braindrane is Fellow and Tutor in Computational Theology at Christnose College, Oxford.

‡Bernard Sufrin was Fellow and Tutor in Computation at Worcester College, Oxford from 1982 until 2012 when he retired. He is currently Tutor in Computer Science at Magdalen College. The Bernard Sufrin Fellowship in Computer Science at Worcester College was established in 2018.

Contents

1 Demiliterate programming

TEXTRACT has a shallower learning-curve than most literate programming systems we have seen; partly because there isn't much to learn. A **TEXTRACT** manuscript is a TEX or LATEX manuscript in which some sections of program code are marked-up *both* to be recognised by the **TEXTRACT** system *and* to be typeset by a TEX or LATEX processor.

TEXTRACT [2] is used to extract program text files from a TEX or LATEX manuscript; this step corresponds to the "tangling" phase described by Knuth and his followers. There is no "weaving" phase at all: the manuscript is typeset by the usual TEX or LATEX processor. As it happens we use the enormously versatile `listings`[3] package to define code section markup in LATEX, but you don't need to if you have something better; and we have recently adapted **TEXTRACT** for use with the differently-versatile `fancyvrb` (see the appendix).

We call this demiliterate programming because the markup and code extraction details are less than half as hard to master as those of most extant literate-programming systems. On the other hand, it cannot compensate for weaknesses in the design of the declarative structures of a programming language: something that many literate-programming systems (try to) achieve.

In short, if you are already happy being a literate programmer then you probably don't need **TEXTRACT**; and if you are a semi-literate programmer there's probably not a lot it can do for you.

TEXTRACT code sections take one of the forms:

```
\begin{kind}                          \begin{kind}[path]
    body                                  body
\end{kind}                            \end{kind}

\begin{kind}···other parameters···    \begin{kind}[path]···other parameters···
    body                                  body
\end{kind}                            \end{kind}
```

The code section's *kind* matches the regular expression: `[+-=*|]*code[+-=*|]*`.[4] The optional *path* parameter (when it appears) is either a *code-snippet* label or a path in the filestore. A code snippet label is a sequence of non-`]` characters that starts with 3 dots or a sequence of non-`'` characters between double-`''quotes''`.

TEXTRACT pays no attention to any *other parameters* that appear.[5] It simply appends the *body* of each code section to the file or snippet specified by its [*path*] parameter. If the particular path is appearing for the first time in a manuscript then the file or snippet it specifies is created first. If the parameter is not present (or blank), then the most recent *path* is used instead.

Lines that consist entirely of snippet labels, *viz* lines of the form `...`*text* or "*text*", are expanded when code bodies are being copied to files.

That's all!

---

2 Getting Started

Here's a TEXTRACT LaTeX manuscript:

```
\documentclass{article}
\usepackage{textract}
\begin{document}
\codestyle{java}
Here's  the \TexTract ``Hello World'' in Java.
\begin{code}[HelloWorld.java]
 public class HelloWorld
 { public static void main(String[] args)
   { System.out.println("Hello World") }
 }
\end{code}
\end{document}
```

and here (to the right of the verticals) is what it looks like when it is LaTeXeD.

Here's the TEXTRACT "Hello World" in Java.

```
public class HelloWorld
{ public static void main(String[] args)
  { System.out.println("Hello World") }
}
```

When it's TEXTRACTeD the code is copied to `HelloWorld.java`.

```
// 294
 public class HelloWorld
 { public static void main(String[] args)
   { System.out.println("Hello World") }
 }
```

The comment on the line before the class declaration indicates the line number – in the latex source file of *this documentation* – of the `begin{code}` declaration that gave rise to the code. Such comments needn't necessarily be present (see the FAQ for details), but we have enabled them for the purposes of generating this documentation.

We expect you already know what this code does after it is compiled and run.

So far so good? But maybe you want to typeset your code in a slightly different form – perhaps as a framed floating listing. The code markup environments we are using for this article are based on the `listings` package. This package offers a good deal of stylistic flexibility to the literate programmer, and does so without the slightest pain save that involved in making good æsthetic choices. You don't have to use it if you don't want to, but it's very well documented and seems very reliable.

We generated Listing 1 by using the following markup.

```
\begin{code+}[Hi.java]{frame=tBlr,float=b,label=HWJ,caption={Hello Chum}}
 public class Hi
 {
   public static void main(String[] args)
   { try
     { System.out.print("Hello"); }
     finally
     { System.out.println("Chum"); }
   }
 }
\end{code+}
```

The `code+` environment takes an additional parameter that specifies further arguments for the `listings` macros. Here we specify a frame, then a floating constraint, a label and a caption: the latter work in the same way as in the `figure` environment.

The resulting code in `Hi.java` is

```
// 364
     public class Hi
     {
       public static void main(String[] args)
       { try
         { System.out.print("Hello"); }
         finally
         { System.out.println("Chum"); }
       }
     }
```

```
public class Hi
{
  public static void main(String[] args)
  { try
    { System.out.print("Hello"); }
    finally
    { System.out.println("Chum"); }
  }
}
```

Listing 1: Hello Chum

The `listings` package lets us declare that material delimited by certain kinds of comment delimiter be suppressed from the listing. This is convenient if we don't want to disfigure our pretty article about a Java program by including javadoc comments in it.

The `|code|` environment and indented style are defined, at the top of this article, by

```
\lstnewenvironment{|code|}[2][]
{\lstset{ captionpos={b}
        , morecomment=[is]{/**}{*/}
        , style=indented,frame={tBlR},#2,}}
{}
\lstdefinestyle{indented}{xleftmargin=1em,xrightmargin=1em,}
```

So we can achieve a framed and titled effect, and suppress the javadoc comment from the typeset article by using the markup:

```
\begin{|code|}[Hello/Again.java]{title={Hello Again}}
 package Hello;
 /**
    <tt>Hello.Again</tt> is a particularly useless program.
 */
 ... Code of HelloAgain
\end{|code|}
```

This gets typeset as:

```
package Hello;

... Code of HelloAgain
```

Hello Again

Notice that no listing number is generated: this is what happens when a code section is `titled` rather than `captioned`.

Among the host of useful features offered by `listings` is the power to typeset code using more mathematical-looking symbols. One way of doing this systematically is to make a supplementary language or style declaration.

For example, the following defines a "mathematical" style:

```
\lstdefinestyle{mathematical}{literate={<-}{$\in$}1{+=}{$:\cup$}2}
```

This can be used as a local style modifier, for example in

```
\begin{code+}[litexample]{style=mathematical}
     for (arg<-args) set += arg
\end{code+}
```

which typesets as:

```
for (arg∈args) set :∪ arg
```

To use it systematically we would incorporate it into a standard code section declaration, for example

```
\lstnewenvironment{+code}[1][]{\lstset{style=mathematical,}}{}
```

---

3 Code Snippets

We imagine that if you got this far you may want to write about more complicated programs than we have so far shown. For example, suppose you want to present a Scala program that imports some library material, then defines a simple class and a main program.

If you want to present it "top-down", you could define its Scala text as follows in the manuscript file:

```
\codestyle{scala}
\begin{code}[TexTract/program.scala]
 package TexTract
 ... The imports
 ... The definition of PowerSet

 object program
 { def main(args: Array[String]) =
   { val set = new PowerSet
     ... The main loop
   }
 }
\end{code}
```

This would be shown in your article as:

```
package TexTract
··· The imports
··· The definition of PowerSet

object program
{ def main(args: Array[String]) =
  { val set = new PowerSet
    ··· The main loop
  }
}
```

You might want to present the body of the program first. It doesn't take much explanation, so you can present it all at once:

```
    for (arg←args) set add arg
    println(set·getSubsets)
```

The listing above was generated by

```
\begin{code}[...The main loop]
     for (arg<-args) set add arg
     println(set.getSubsets)
\end{code}
```

Notice that the name of the code section starts with three dots. The effect of this is to define a chunk of program text (a so-called *code-snippet*) with the symbolic label: "themainloop".[6] When the body of a code section is finally output to its file, each

line that consists only of a reference to a snippet, *i.e.* each line of the form `...`*text* or "*text*", is replaced by the body of the snippet (if it has been defined), and replaced by the text (marked as a comment) otherwise.

To define the imports snippet we might write

```
\begin{|code|}[... The imports]{title={The imports}}
 import  scala.collection.mutable._
 import  java.io._
 import  java.util.regex._
\end{|code|}
```

and it would be typeset as:

```
import  scala·collection·mutable··
import  java·io··
import  java·util·regex··
```

The imports

If we forget to include a code section describing "The definition of Powerset", then the code eventually written to `program.scala` will be:

```
// 486
 package TexTract
// 549
 import  scala.collection.mutable._
 import  java.io._
 import  java.util.regex._
 //   The definition of PowerSet
// 489

 object program
 { def main(args: Array[String]) =
   { val set = new PowerSet
// 527
     for (arg<-args) set add arg
     println(set.getSubsets)
// 494
   }
 }
```

Notice that: "`...  The definition of Powerset`" has been transformed into a one-line comment.

---

4 Special Code Sections

4.1 Anonymous Sections

When a code section without a name appears, the enclosed code is typeset in the usual way, but it is appended to the previously-named code section or file.

This makes it convenient to intercalate sections of code intended for the same file or section between latex comments.

4.2 Hidden Sections

The standard TEXTRACT style file defines the `code*` section to treat its content as a LaTeX comment. Material within such sections is still extracted in the usual way.

A simple example of a passage that uses hidden and anonymous sections is:

```
\begin{code*}[What/Ho.java]
package What;
\end{code*}
This is the code of the ‘‘What.Ho’’ program.
As you can see, it is not very complicated:
\begin{code}
public static void main(String[] args)
{
  System.err.println("What ");
\end{code}
and we don’t need to do a sophisticated proof of correctness.
\begin{code}
  System.err.println("Ho?");
}
\end{code}
```

In the java code style this gets typeset as:

> This is the code of the “What.Ho” program. As you can see, it is not very complicated:
>
> ```
> public static void main(String[] args)
> {
>   System.err.println("What ");
> ```
>
> and we don’t need to do a sophisticated proof of correctness.
>
> ```
>   System.err.println("Ho?");
> }
> ```

The resulting code is

```
// 608
package What;
// 613
public static void main(String[] args)
{
  System.err.println("What ");
// 619
  System.err.println("Ho?");
}
```

---

5 Tips and Tricks

1. Find out as much as you can about the `listings` package: in particular how to define language dialects, styles and new listing environments.

2. Use a fairly small set of `code`-like environments in your documents.

3. If you want to typeset a code section without it being sent to a file, the best thing to do is to define a listing environment whose name doesn't match the code-section pattern. Short of that, a good hack is to use the snippet with the empty name as the path part of an ordinary code section. For example (and just to show what `listings` makes of Haskell), the code section:

```
\begin{code+}[...]{language=haskell,style=mathematical}
rats :: [ Rational ]
rats  = concat (diags[[m/n | m<-[1..]] | n<-[1..]])
diags = diags' []
        where
         diags' xss (ys:yss) =
                map head xss : diags'(ys:map tail xss) yss
\end{code+}
```

gets typeset as:

```
rats :: [ Rational ]
rats  = concat (diags[[m/n | m∈[1..]] | n∈[1..]])
diags = diags' []
        where
         diags' xss (ys:yss) =
                map head xss : diags'(ys:map tail xss) yss
```

Appendix

A FAQ

Q: Can I use more than one programming language in a single TEXTRACT manuscript?

A: Yes; and you can also generate more than one program file from a single TEXTRACT manuscript.

Q: Must the whole of a code section macro call appear on a single line in the manuscript?

A: It must; and we certainly won't be writing a LaTeX or TeX parser to change that.

Q: How hard is tracking error message locations?

A: Unless you use snippets or output compression it's trivial: output files are written so that manuscript and output file line numbers are identical. If you set `number=on` in the `<textract>` ant task, then compression is enabled and output files are annotated with source manuscript line numbers. Each annotation is in the form of a line-number pragma for the language being output if TEXTRACT knows that form; otherwise it's in the form of a comment for that language. For example, these code blocks – embedded in the latex source file of *this documentation*:

```
\begin{code}[propositions.hs]
module Propositions where
  ... Imports
  data Prop = Atom Atomic         -- a, b ...
            | Not  Prop           -- not p
            | Prop `And` Prop     -- p ∧ q
            | Prop `Or`  Prop     -- p ∨ q
            | Prop `Imp` Prop     -- p ⇒ q
            | Prop `Iff` Prop     -- p ⇔ q
            deriving (Eq)

  instance (Show Prop) where showsPrec = showProp
\end{code}

%
% The main story about propositions
%

\begin{code}[... Imports]
import Prelude(Eq, String, Show, ShowS, Int,
               map, (.), (++), foldr, take,
               unlines, (>=), (+), (*), ($),
               showParen, showString, showsPrec, show)
\end{code}
```

Give rise to the Haskell source file `propositions.hs`)

```
{-# LINE 724 "/Users/sufrin/GitHomes/TexTract/doc/textract.tex" #-}
      module Propositions where
{-# LINE 742 "/Users/sufrin/GitHomes/TexTract/doc/textract.tex" #-}
```

```
      import Prelude(Eq, String, Show, ShowS, Int,
                     map, (.), (++), foldr, take,
                     unlines, (>=), (+), (*), ($),
                     showParen, showString, showsPrec, show)
{-# LINE 726 "/Users/sufrin/GitHomes/TexTract/doc/textract.tex" #-}
        data Prop = Atom Atomic         -- a, b ...
                  | Not  Prop           -- not p
                  | Prop `And` Prop     -- p ∧ q
                  | Prop `Or`  Prop     -- p ∨ q
                  | Prop `Imp` Prop     -- p ⇒ q
                  | Prop `Iff` Prop     -- p ⇔ q
                  deriving (Eq)

        instance (Show Prop) where showsPrec = showProp
```

The answer to the next question may also be helpful.

Q: Is snippet-substitution nested?

A: Judge for yourself. This code block:

```
\begin{code}[foo.tex]
... Undefined snippet
... Answer
\end{code}
\begin{code}[... first clause]
Snippet-expansion takes place in the final output
phase of \TexTract,
\end{code}
\begin{code}[... Answer]
... first clause
and the bodies of nested snippets are themselves
subject to snippet-expansion as they are output.
... another undefined snippet
\end{code}
```

leads to this output (in `foo.tex`)

```
     %       Undefined snippet
% 767
% 763
     Snippet-expansion takes place in the final output
     phase of \TexTract,
% 768
     and the bodies of nested snippets are themselves
     subject to snippet-expansion as they are output.
     %       another undefined snippet
```

The first line-number annotation is the line number – in the latex source of this document – of the (snippet-call) text `... Undefined snippet`. The next three annotations denote, respectively, the line number of the (snippet-call) text `... Answer`, the line number of the (snippet-call) `... first clause`, and the line number of the first line of the expansion of `... first clause`.

B Tools

B.1 Ant Task

To define the ant task in your build task:

```
<taskdef name="textract"
         classpath="${TEXTRACTHOME}/textract.jar" classname="ant.Textract"/>
```

The task has the following parameters

			DEFAULT	Effect of default setting
<textract	root	=	"TEXTRACT"	Output paths are prefixed with this path
	compress	=	"off"	Output files are not compressed
	code	=	"code"	core of the block kind of a code section
	ans	=	"ans|answer"	pattern matching the block kind of an answer environment
	answers	=	"on"	code within ans(wer) environments is not excluded
	number	=	"off"	No line-number comments are placed in output files
	enc	=	"UTF8"	Encoding of input and output files
	force	=	"off"	If source is up-to-date output is not regenerated
	errnone	=	"on"	Fail if no existing source files are specified
	legacy	=	"off"	Legacy processing is disabled (see below)
	code	=	"code"	Specify the {code} pattern
	comments	=	"|.hs|{-|-}|.java|//||.scala|//||.tex|%||"	Specify comment conventions for output files
	pragmas	=	"|.hs|{-# LINE %n %f¨ #-}|.c|#line %n %f¨"	Specify line number pragma conventions for output files

It also takes a `file=`*`path`* to specify a single file to process, or one or more nested `<fileset>...</fileset>` elements to specify a collection of existing files. It is usually an error if such a specification results in no files to process.

When output files are compressed there is no attempt to correlate the location of an output line to its location in the source manuscript.

Comment conventions have to be specified all at once. The specification takes the form $\underline{\oplus}spec\underline{\oplus}spec\underline{\oplus}spec...$ where $\underline{\oplus}$ is any character that is not going to be part of a comment specification, and all *spec*s take the form $.ext\underline{\oplus}left\underline{\oplus}right$, where $left$ and *right* are $\underline{\oplus}$-free strings denoting the opening (respective closing) brackets of a comment, and *ext* is the filename extension.

Line-number pragma conventions are specified similarly. Each *spec* takes the form $.ext\underline{\oplus}format$, where $format$ is a $\underline{\oplus}$-free string. When outputting a line-number annotation, this text is written with `%f` replaced (everywhere) by the input filename, and `%n` replaced (everywhere) by the line number.

B.2 Command-Line Task

```
textract [switch]* [file]*

  -c                compress output (eliminate latex source lines)
  -a                suppress classes contained within {ans|answer} environments (implies -c)
  -d=<root>         specify root directory for output
  -c=<root>         implies -c -d=<root>
  -a=<root>         implies -a -c=<root>
  -f                force code generation, even if the generated files are up to date
  -l                force legacy code block processing
  -n                place line-number comments on output lines
  -enc=<enc>        Input and output file(s) are in the given encoding (default is UTF8)
  -code=<pat>       Specify the {code} pattern -- initially 'code'
  -ans=<specs>      Specify the {ans} pattern -- initially ans|answer
  -comments=<specs> Specify output language comment conventions
                    (see Ant Task)
  -pragmas=<specs>  Specify output language line number pragma conventions
                    (see Ant Task)
```

This assumes that the `textract` command is the equivalent of

```
scala -cp $TEXTRACTHOME/textract.jar textract
```

C textract.sty

```
%
% Designed to work with TexTract to weave code from latex sources
%
% $Id: textract.sty 61 2010-04-26 10:21:55Z sufrin $
%
\ProvidesPackage{textract}
\RequirePackage{breakverbatim}
\RequirePackage{listings}
\RequirePackage{ifthen}

%
% Style-definition and style switching
%
\def\newcodestyle#1#2{\lstdefinestyle{#1}{language=#1,#2,}}
\def\codestyle#1{\lstset{style=#1}}

%
% A few styles
%
\newcodestyle{java}
 {basicstyle=\classsize\sffamily,}

\newcodestyle{scala}
 {basicstyle=\classsize\sffamily,xleftmargin=1em,xrightmargin=1em,}

%\newcodestyle{haskell}
% {basicstyle=\classsize\sffamily,xleftmargin=1em,xrightmargin=1em,}

\newcodestyle{xml}
 {basicstyle=\classsize\ttfamily,xleftmargin=1em,xrightmargin=1em,}

%
% Default style is Java
%
\codestyle{java}

%
% {code} blocks are typeset in the standard form and extracted by TexTract
%
\lstnewenvironment{code}[1][]
{\lstset{xleftmargin=1em,xrightmargin=1em,}}
{}
%
% {code+} blocks are typeset with additional formatting and extracted by TexTract
%
\lstnewenvironment{code+}[2][]
{\lstset{xleftmargin=1em,xrightmargin=1em,captionpos={b},#2,}}
{}
%
% {code*} blocks are treated as comments and extracted by TexTract
%
\expandafter\def\csname code*\endcsname#1{\comment}
\expandafter\let\csname endcode*\endcsname=\endcomment

%
%
%
\def\unquote``#1''{\textrm{#1}}
\def\textcode#1{{\ttfamily\lstinline{#1}}}
\def\textcodett#1{{\sffamily\lstinline{#1}}}
```

```
%
%
%
\ifx \smaller\undefined \def\smaller{\normalsize} \fi
\ifx \xmlsize\undefined \def\classsize{\smaller} \fi
\ifx \classsize\undefined \def\classsize{\smaller} \fi
\ifx \javasize\undefined \def\javasize{\smaller} \fi
\ifx \Javasize\undefined \def\Javasize{} \fi
\ifx \smalljavasize\undefined \def\smalljavasize{\normalsize} \fi

%
% Doesn't do anything, but doc blocks are transformed into java/scala block comments.

%
\newenvironment{doc}[2][]{\begin{smaller}}{\end{smaller}}
```

D Legacy Processing

In legacy processing mode many bets are off. Code blocks take one of the forms

```
\begin{class}{java filename without .java extension}
  body
\end{class}

\begin{hideclass}{java filename without .java extension}
  body
\end{hideclass}

\begin{obj}{scala filename without .scala extension}
  body
\end{obj}

\begin{hideobj}{scala filename without .scala extension}
  body
\end{hideobj}
```

E Using fancyvrb instead of listings

TeXTeRACT has been adapted to the recently-released (2020) fancyvrb style, which supports (the definition of) a host of very sophisticated verbatim-like environments that do not provide for the decoration of programming language keywords, but are completely compatible (as listings is not) with UTF8 character encodings. To use TeXTeRACT with fancyvrb and UTF8 you should

```
\usepackage{fancyvrb}
\usepackage[mathletters]{ucs}
\usepackage[utf8x]{inputenc}
```

Then, later, for as many different code-style environments as you wish to define, use the appropriate fancyvrb declaration, for example:

```
\DefineVerbatimEnvironment{code}{Verbatim}{fontsize=\small}
```

and somewhere before the code-like environment(s) you wish to save as files, place a comment of the form

```
%%%%%%%%%%%%%%%%%%%% [path or code snippet label]  ... anything
```

This serves to set the *path* or snippet-label for subsequent code-like environments. When this form of path or snippet-label specification is used, the optional parameter following it is ignored by TeXTeRACT so that optional parameters to the fancyvrb-style environments can be interpreted properly, *i.e* as *customization specifications*. For example,

```
%%%%%%%%%% [propositions.hs]
\begin{code}[frame=single,label=Haskell]
  data Prop = Atom Atomic         -- a, b ...
            | Not  Prop           -- not p
            | Prop `And` Prop     -- p ∧ q
            | Prop `Or`  Prop     -- p ∨ q
            | Prop `Imp` Prop     -- p ⇒ q
            | Prop `Iff` Prop     -- p ⇔ q
            deriving (Eq)
\end{code}
```

Notes

1. Anne R. Surfbird: *On seeing the first literate programming system of Spring*, Onomatopœia Inc., March 2008. (Quoted without permission)

2. The $\TeX$ system can be invoked as a command-line program or an Ant task.

3. `http://www.ctan.org/tex-archive/macros/latex/contrib/listings/`

4. In other words a code section's *kind* is a sequence of characters consisting of zero or more occurences of one of the characters `"+-=*|"`, followed by the word `"code"`, followed by zero or more occurences of one of the characters `"+-=*|"`. In both the Ant task and the command-line program even the word `"code"\`can be changed to an arbitrary regular expression pattern by setting a parameter. For example, if you wanted your "code" environments to be named either `CODE` or `LOGIC`, you could set `code="CODE|LOGIC"` (see appendix B).

5. The standard $\TeX$ style file specifies a few specific *kind*s, each with its own fixed number of (non-optional) parameters.

6. To avoid hard-so-spot mistakes in references to them, snippet labels are "normalized" by removing any spaces, and transforming their letters to lowercase.

GitHub Version 5 of Thu Aug 26 14:55:16 2021 +0100
(Formerly Subversion Revision: 83)